Exhibit 8

Exhibit 10: U.S. Patent No. 7,372,960

Claim 3	Exemplary Evidence of Infringement
3[pre] A method of performing a finite field operation on elements of a finite field, comprising the steps of:	Core Scientific, Inc. (hereinafter "Core") performs a method for finite field operation on elements of a finite field, for example, during the transfer of Bitcoin to an address, which is a cryptographic operation. *See*, *e.g.*: "The Company sells bitcoin it receives through mining…. Sales of digital assets awarded to the Company through its self-mining activities are classified as cash flows from operating activities. The Company does not have any off-balance sheet holdings of digital assets and does not safeguard digital assets for third parties." *See*, *e.g.*, Core Scientific., Inc., Quarterly report pursuant to Section 13 and 15(d), (Form 10-Q), at Note 1, filed Nov. 06, 2024, available at https://www.sec.gov/ix?doc=/Archives/edgar/data/1839341/000162828024045811/core-20240930.htm "Core Scientific, Inc. is a leader in digital infrastructure for bitcoin mining and high-performance computing. We operate dedicated, purpose-built facilities for digital asset mining and are a premier provider of digital infrastructure, software solutions and services to our third-party customers. We employ are own large fleet of computers ('miners') to earn digital assets for our own account and we provide hosting services for large bitcoin mining customers …. We derive the majority of our revenue from earning bitcoin for our own account ('self-mining')." *See*, *e.g.*, Core Scientific., Inc., Quarterly report pursuant to Section 13 and 15(d), (Form 10-Q), at Note 1, filed Nov. 06, 2024, available at https://www.sec.gov/ix?doc=/Archives/edgar/data/1839341/000162828024045811/core-20240930.htm We finance our operations primarily through cash generated from operations, including the sale of self-mined bitcoin …." *See*, *e.g.*, Core Scientific, Inc. Form 10-K, at 67, filed Feb. 27, 2025, available at https://investors.corescientific.com/sec-filings/all-sec-filings/content/0001628280-25-008302/0001628280-25-008302.pdf

Claim 3	Exemplary Evidence of Infringement

	"**Bitcoin signed messages have three parts, which are the Message, Address, and Signature**. The message is the actual message text - all kinds of text is supported, but it is recommended to avoid using non-ASCII characters in the signature because they might be encoded in different character sets, preventing signature verification from succeeding.
	
	The address is a legacy, nested segwit, or native segwit address. Message signing from legacy addresses was added by Satoshi himself and therefore does not have a BIP. **Message signing from segwit addresses has been added by BIP137 … The Signature is a base64-encoded ECDSA signature** that, when decoded, with fields described in the next section." (Emphasis added)
	
	See, *e.g.,* Message Signing, https://en.bitcoin.it/wiki/Message_signing.
	
	"This document describes a signature format for **signing messages with Bitcoin private keys**.
	
	The specification is intended to describe the standard for signatures of messages that can be signed and verified between different clients that exist in the field today." (Emphasis added)
	
	See, *e.g.*, Bitcoin BIP137, https://github.com/bitcoin/bips/blob/master/bip-0137.mediawiki.
	
	For example, in secp256k1, type secp256k1_fe consists of 5 or 10 machine words, depending on the machine's word size: "field_5x52.h" applies to machines with 64bit word size and "field_10x32.h" applies to machines with 32bit word size. *See*, *e.g.*:
	
	`/** This field implementation represents the value as 5 uint64_t limbs in base`
	` * 2^52. */`
	`typedef struct {`
	`   /* A field element f represents the sum(i=0..4, f.n[i] << (i*52)) mod p,`
	`    * where p is the field modulus, 2^256 - 2^32 - 977.`
	`    *`
	`    * The individual limbs f.n[i] can exceed 2^52; the field's magnitude roughly`
	`    * corresponds to how much excess is allowed. The value`
	`    * sum(i=0..4, f.n[i] << (i*52)) may exceed p, unless the field element is`
	`    * normalized. */`
	`    uint64_t n[5];`

Claim 3	Exemplary Evidence of Infringement

	<pre> /*
	     * Magnitude m requires:
	     * n[i] <= 2 * m * (2^52 - 1) for i=0..3
	     * n[4] <= 2 * m * (2^48 - 1)
	     *
	     * Normalized requires:
	     * n[i] <= (2^52 - 1) for i=0..3
	     * sum(i=0..4, n[i] << (i*52)) < p
	     * (together these imply n[4] <= 2^48 - 1)
	     */
	    SECP256K1_FE_VERIFY_FIELDS
	} secp256k1_fe;</pre>
	See, *e.g.*, bitcoin/src/secp256k1/src/field_5x52.h
	
	"The points on the elliptic curve are the pairs of finite field elements."
	See, *e.g.*, '960 pat. at col. 1, lines 45-47.
	
	<pre>/** A group element in affine coordinates on the secp256k1 curve,
	 * or occasionally on an isomorphic curve of the form y^2 = x^3 + 7*t^6.
	 * ...
	 */
	typedef struct {
	    secp256k1_fe x;
	    secp256k1_fe y;
	    int infinity; /* whether this represents the point at infinity */
	} secp256k1_ge;
	
	...
	
	/** A group element of the secp256k1 curve, in jacobian coordinates.
	 * ...
	 */
	typedef struct {
	    secp256k1_fe x; /* actual X: x/z^2 */
	    secp256k1_fe y; /* actual Y: y/z^3 */
	    secp256k1_fe z;
	    int infinity; /* whether this represents the point at infinity */
	} secp256k1_gej;</pre>

<table>
<tr><th>Claim 3</th><th>Exemplary Evidence of Infringement</th></tr>
<tr><td></td><td>See, e.g., bitcoin/src/secp256k1/src/group.h</td></tr>
<tr><td>3[a] a) representing each element as a predetermined number of machine words;</td><td>Core represents each element as a predetermined number of machine words.

For example, Core’s miners represent each element as a predetermined number of machine words.

For example, in secp256k1, type secp256k1_fe consists of 5 or 10 machine words, depending on the machine’s word size: “field_5x52.h” applies to machines with 64bit word size and “field_10x32.h” applies to machines with 32bit word size. See, e.g.:

<pre>
/** This field implementation represents the value as 5 uint64_t limbs in base
 * 2^52. */
typedef struct {
   /* A field element f represents the sum(i=0..4, f.n[i] << (i*52)) mod p,
    * where p is the field modulus, 2^256 - 2^32 - 977.
    *
    * The individual limbs f.n[i] can exceed 2^52; the field's magnitude roughly
    * corresponds to how much excess is allowed. The value
    * sum(i=0..4, f.n[i] << (i*52)) may exceed p, unless the field element is
    * normalized. */
    uint64_t n[5];
    /*
     * Magnitude m requires:
     * n[i] <= 2 * m * (2^52 - 1) for i=0..3
     * n[4] <= 2 * m * (2^48 - 1)
     *
     * Normalized requires:
     * n[i] <= (2^52 - 1) for i=0..3
     * sum(i=0..4, n[i] << (i*52)) < p
     * (together these imply n[4] <= 2^48 - 1)
     */
    SECP256K1_FE_VERIFY_FIELDS
} secp256k1_fe;
</pre>
See, e.g., bitcoin/src/secp256k1/src/field_5x52.h</td></tr>
</table>

<table>
<tr><th>Claim 3</th><th>Exemplary Evidence of Infringement</th></tr>
<tr><td></td><td>“The points on the elliptic curve are the pairs of finite field elements.”
See, *e.g.*, ’960 pat. at col. 1, lines 45-47.
<pre>/** A group element in affine coordinates on the secp256k1 curve,
 * or occasionally on an isomorphic curve of the form y^2 = x^3 + 7*t^6.
 * ...
 */
typedef struct {
    secp256k1_fe x;
    secp256k1_fe y;
    int infinity; /* whether this represents the point at infinity */
} secp256k1_ge;

...

/** A group element of the secp256k1 curve, in jacobian coordinates.
 * ...
 */
typedef struct {
    secp256k1_fe x; /* actual X: x/z^2 */
    secp256k1_fe y; /* actual Y: y/z^3 */
    secp256k1_fe z;
    int infinity; /* whether this represents the point at infinity */
} secp256k1_gej;</pre>
See, *e.g.*, bitcoin/src/secp256k1/src/group.h</td></tr>
<tr><td>**3[b]** b) performing a non-reducing wordsized operation on said representations, said wordsized operation corresponding to said finite field operation;</td><td>Core performs a non-reducing wordsized operation on said representations, said wordsized operation corresponding to said finite field operation

For example, Core’s miners perform a non-reducing wordsized operation (*e.g.*, executing `secp256k1_ge_set_gej`) on said representations, said wordsized operation corresponding to said finite field operation.

For example, the result of `secp256k1_ge_set_gej` is unreduced and needs normalizing. *See*, *e.g.*:</td></tr>
</table>

<table>
<tr><th>Claim 3</th><th>Exemplary Evidence of Infringement</th></tr>
<tr><td></td><td>

```
static int secp256k1_ecdsa_sig_sign(const secp256k1_ecmult_gen_context *ctx,
  secp256k1_scalar *sigr, secp256k1_scalar *sigs, const secp256k1_scalar *seckey,
  const secp256k1_scalar *message, const secp256k1_scalar *nonce, int *recid
) {
    ...; secp256k1_ge r; ...;
    secp256k1_ecmult_gen(ctx, &rp, nonce);
    secp256k1_ge_set_gej(&r, &rp);
    secp256k1_fe_normalize(&r.x);
    secp256k1_fe_normalize(&r.y);
    secp256k1_fe_get_b32(b, &r.x);
    secp256k1_scalar_set_b32(sigr, b, &overflow);
    ...; if (...) { *recid = (...) | secp256k1_fe_is_odd(&r.y); } ...;
}
```

See, *e.g.*, bitcoin/src/secp256k1/src/ecdsa_impl.h

The function `secp256k1_ge_set_gej` invokes `secp256k1_fe_mul`. That function invokes `secp256k1_fe_impl_mul`. That function invokes `secp256k1_fe_mul_inner`. Function `secp256k1_u128_accum_mul` is then invoked for each word of above finite field element **r** (typed `secp256k1_ge`). *See*, *e.g.*:

```
/* Multiply two unsigned 64-bit values a and b and write the result to r. */
static SECP256K1_INLINE void secp256k1_u128_mul(secp256k1_uint128 *r,
  uint64_t a, uint64_t b);
```

See, *e.g.*, bitcoin/src/secp256k1/src/int128.h
</td></tr>
<tr><td>**3[c]** c) completing said non-reducing wordsized operation for each word of said representations to obtain an unreduced result;</td><td>

Core complets said non-reducing wordsized operation for each word of said representations to obtain an unreduced result.

For example, Core's miners complete said non-reducing wordsized operation (*e.g.*, executing `secp256k1_ge_set_gej`) for each word of said representations to obtain an unreduced result.

For example, the result of `secp256k1_ge_set_gej` is unreduced and needs normalizing. *See*, *e.g.*:

```
static int secp256k1_ecdsa_sig_sign(const secp256k1_ecmult_gen_context *ctx,
```
</td></tr>
</table>

<table>
<tr><th>Claim 3</th><th>Exemplary Evidence of Infringement</th></tr>
<tr><td></td><td>

```
  secp256k1_scalar *sigr, secp256k1_scalar *sigs, const secp256k1_scalar *seckey,
  const secp256k1_scalar *message, const secp256k1_scalar *nonce, int *recid
) {
    ...; secp256k1_ge r; ...;
    secp256k1_ecmult_gen(ctx, &rp, nonce);
    secp256k1_ge_set_gej(&r, &rp);
    secp256k1_fe_normalize(&r.x);
    secp256k1_fe_normalize(&r.y);
    secp256k1_fe_get_b32(b, &r.x);
    secp256k1_scalar_set_b32(sigr, b, &overflow);
    ...; if (...) { *recid = (...) | secp256k1_fe_is_odd(&r.y); } ...;
}
```

See, *e.g.*, bitcoin/src/secp256k1/src/ecdsa_impl.h

The function `secp256k1_ge_set_gej` invokes `secp256k1_fe_mul`. That function invokes `secp256k1_fe_impl_mul`. That function invokes `secp256k1_fe_mul_inner`. Function `secp256k1_u128_accum_mul` is then invoked for each word of above finite field element **r** (typed `secp256k1_ge`). *See*, *e.g.*:

```
/* Multiply two unsigned 64-bit values a and b and write the result to r. */
static SECP256K1_INLINE void secp256k1_u128_mul(secp256k1_uint128 *r,
  uint64_t a, uint64_t b);
```

See, *e.g.*, bitcoin/src/secp256k1/src/int128.h</td></tr>
<tr><td>3[d] d) upon computing said unreduced result, performing a specific modular reduction of said unreduced result to reduce said unreduced result to that of a field element of said finite field to obtain a reduced result; and</td><td>Core, upon computing said unreduced result, performs a specific modular reduction of said unreduced result to reduce said unreduced result to that of a field element of said finite field to obtain a reduced result.

For example, Core's miners, upon computing said unreduced result, perform a specific modular reduction (e.g., <code>secp256k1_fe_normalize</code>) of said unreduced result to reduce said unreduced result to that of a field element of said finite field to obtain a reduced result.

For example, the result of <code>secp256k1_ge_set_gej</code> is unreduced and needs normalizing. See, e.g.:</td></tr>
</table>

Claim 3	Exemplary Evidence of Infringement

	<pre>static int secp256k1_ecdsa_sig_sign(const secp256k1_ecmult_gen_context *ctx,
	  secp256k1_scalar *sigr, secp256k1_scalar *sigs, const secp256k1_scalar *seckey,
	  const secp256k1_scalar *message, const secp256k1_scalar *nonce, int *recid
	) {
	    ...; secp256k1_ge r; ...;
	    secp256k1_ecmult_gen(ctx, &rp, nonce);
	    secp256k1_ge_set_gej(&r, &rp);
	    secp256k1_fe_normalize(&r.x);
	    secp256k1_fe_normalize(&r.y);
	    secp256k1_fe_get_b32(b, &r.x);
	    secp256k1_scalar_set_b32(sigr, b, &overflow);
	    ...; if (...) { *recid = (...) | secp256k1_fe_is_odd(&r.y); } ...;
	}</pre> *See*, *e.g.*, bitcoin/src/secp256k1/src/ecdsa_impl.h
	
	The function `secp256k1_ge_set_gej` invokes `secp256k1_fe_mul`. That function invokes `secp256k1_fe_impl_mul`. That function invokes `secp256k1_fe_mul_inner`. Function `secp256k1_u128_accum_mul` is then invoked for each word of above finite field element **r** (typed `secp256k1_ge`). *See*, *e.g.*:<pre>/* Multiply two unsigned 64-bit values a and b and write the result to r. */
	static SECP256K1_INLINE void secp256k1_u128_mul(secp256k1_uint128 *r,
	  uint64_t a, uint64_t b);</pre> *See*, *e.g.*, bitcoin/src/secp256k1/src/int128.h
	
	The function `secp256k1_fe_impl_normalize` ensures a field element does not exceed "`field modulus, 2^256 - 2^32 - 977`", per "`field_5x52.h`". *See*, *e.g.*:<pre>SECP256K1_INLINE static void secp256k1_fe_normalize(secp256k1_fe *r) {
	    ...; secp256k1_fe_impl_normalize(r); ...;
	}</pre> *See*, *e.g.*, bitcoin/src/secp256k1/src/field_impl.h

<table>
<tr><th>Claim 3</th><th>Exemplary Evidence of Infringement</th></tr>
<tr><td></td><td>

```
static void secp256k1_fe_impl_normalize(secp256k1_fe *r) {
    uint64_t t0 = r->n[0], t1 = r->n[1], t2 = r->n[2], t3 = r->n[3], t4 = r->n[4];
    /* Reduce t4 at the start so there will be at most a single carry from the first
       pass */ ...;
    /* The first pass ensures the magnitude is 1, ... */ ...;
    /* ... except for a possible carry at bit 48 of t4 (i.e. bit 256 of the field
       element) */ ...;
    /* At most a single final reduction is needed; check if the value is >= the
    field characteristic */ ...;
    /* Apply the final reduction (for constant-time behaviour, we do it always) */ ...;
    /* If t4 didn't carry to bit 48 already, then it should have after any final
       reduction */ ...;
     /* Mask off the possible multiple of 2^256 from the final reduction */ ...;
     r->n[0] = t0; r->n[1] = t1; r->n[2] = t2; r->n[3] = t3; r->n[4] = t4;
}
```

See, *e.g.*, bitcoin/src/secp256k1/src/field_5x52_impl.h</td></tr>
<tr><td>3[e] e) using said reduced result in a cryptographic operation.</td><td>Core uses said reduced result in a cryptographic operation.

For example, Core's miners use said reduced result in a cryptographic operation.

For example, the result of <code>secp256k1_ge_set_gej</code> is unreduced and needs normalizing. See, e.g.:

```
static int secp256k1_ecdsa_sig_sign(const secp256k1_ecmult_gen_context *ctx,
  secp256k1_scalar *sigr, secp256k1_scalar *sigs, const secp256k1_scalar *seckey,
  const secp256k1_scalar *message, const secp256k1_scalar *nonce, int *recid
) {
    ...; secp256k1_ge r; ...;
    secp256k1_ecmult_gen(ctx, &rp, nonce);
    secp256k1_ge_set_gej(&r, &rp);
    secp256k1_fe_normalize(&r.x);
    secp256k1_fe_normalize(&r.y);
    secp256k1_fe_get_b32(b, &r.x);
    secp256k1_scalar_set_b32(sigr, b, &overflow);
    ...; if (...) { *recid = (...) | secp256k1_fe_is_odd(&r.y); } ...;
}
```

</td></tr>
</table>

Claim 3	Exemplary Evidence of Infringement

	See, *e.g.*, bitcoin/src/secp256k1/src/ecdsa_impl.h
	
	The function `secp256k1_ge_set_gej` invokes `secp256k1_fe_mul`. That function invokes `secp256k1_fe_impl_mul`. That function invokes `secp256k1_fe_mul_inner`. Function `secp256k1_u128_accum_mul` is then invoked for each word of above finite field element **r** (typed `secp256k1_ge`). *See*, *e.g.*:
	
	`/* Multiply two unsigned 64-bit values a and b and write the result to r. */`
	`static SECP256K1_INLINE void secp256k1_u128_mul(secp256k1_uint128 *r,`
	`  uint64_t a, uint64_t b);`
	
	See, *e.g.*, bitcoin/src/secp256k1/src/int128.h